CS 267 Applications of Parallel Computers

Lecture 12:

Floating Point Arithmetic

David H. Bailey

Based on previous notes by James Demmel and Dave Culler

http://www.nersc.gov/~dhbailey/cs267

Outline

- A little history
- IEEE floating point formats
- Error analysis
- Exception handling
- How to get extra precision cheaply
- Cray arithmetic - a pathological example
- Dangers of parallel and heterogeneous computing

Floating Point Arithmetic: A Little History

- **1947: Von Neumann and Goldstine:**

  "Can't expect to solve most big [n>15] linear systems without carrying many decimal digits, otherwise the computed answer would be completely inaccurate." (wrong)

- **1949: Turing mentions principle of backward error analysis (BEA):**

  "Carrying d digits is equivalent to changing the input data in the d-th place and then solving Ax=b. So if A is only known to d digits, the answer is as accurate as the data deserves."

- **1961: BEA rediscovered and publicized by Wilkinson.**

- **1960s: Numerous papers presented with BEA results for various algorithms.**

- **1960s-1970s: Each computer handled FP arithmetic differently:**
  - **Format, accuracy, rounding mode and exception handling all differed.**
  - **It was very difficult to write portable and reliable technical software.**

- **1982: IEEE-754 standard defined. First implementation: Intel 8087.**

- **1989: ACM Turing Award given to W. Kahan of UCB for design of the IEEE floating point standards.**

- **2000: IEEE FP arithmetic is now almost universally implemented in general purpose computer systems.**

Defining Floating Point Arithmetic

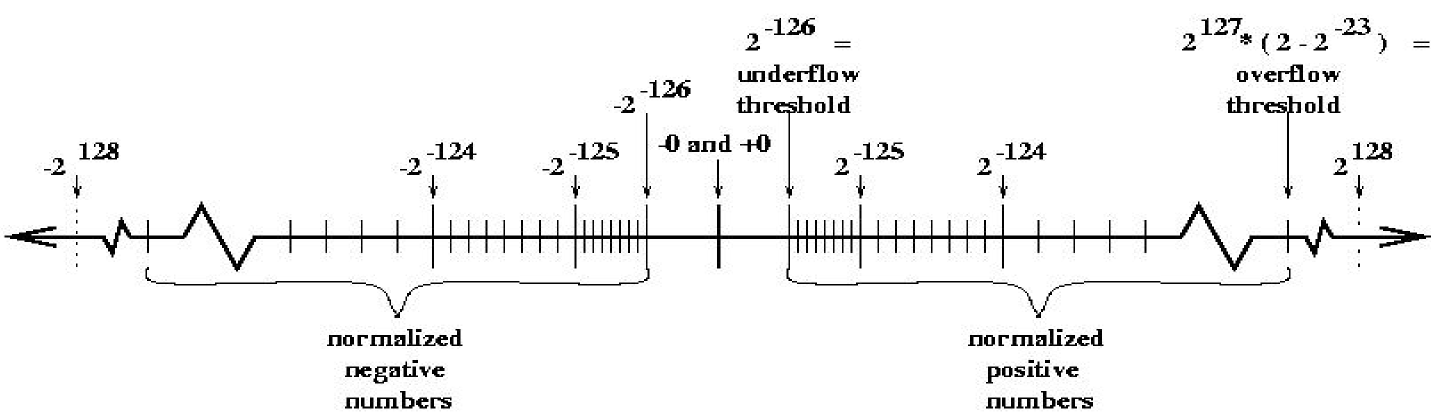
- **Representable numbers:**
  - Scientific notation: +/- d.d…d x r^{exp}
  - sign bit +/-
  - radix r (usually 2 or 10, sometimes 16)
  - significand d.d…d (how many base-r digits d?)
  - exponent exp (range?)
  - others?
- **Operations:**
  - arithmetic: +,-,x,/,... How is result rounded to fit in format?
  - comparison (<, =, >)
  - conversion between different formats - short to long FP numbers, FP to integer, etc.
  - exception handling - what to do for 0/0, 2*largest_number, etc.
  - binary/decimal conversion - for I/O, when radix is not 10.
- **Language/library support is required for all these operations.**

IEEE Floating Point Arithmetic Standard 754 - Normalized Numbers

- **Normalized Nonzero Representable Numbers: +- 1.d…d x 2^{exp}**
  - **Macheps** = Machine epsilon = $2^{-\#\text{significand bits}}$ = relative error in each operation
  - **OV** = overflow threshold = largest number
  - **UN** = underflow threshold = smallest number

Format	# bits	#significand bits	macheps	#exponent bits	exponent range
Single	32	23+1	2^{-24} ($\sim 10^{-7}$)	8	2^{-126} - 2^{127} ($\sim 10^{+-38}$)
Double	64	52+1	2^{-53} ($\sim 10^{-16}$)	11	2^{-1022} - 2^{1023} ($\sim 10^{+-308}$)
Double Extended (*80 bits on all Intel machines*)	>=80	>=64	<=2^{-64}($\sim 10^{-19}$)	>=15	2^{-16382} - 2^{16383} ($\sim 10^{+-4932}$)

- **+- Zero: +-, significand and exponent all zero**
  - **Why bother with -0 later**



IEEE 64-bit Floating-Point Format

- Bit 0: sign of entire number.
- Bits 1-11: exponent, offset by 2^10.
- Bits 12-63: mantissa.
- +0 and -0 are represented differently.
- For normalized nonzero data, a “1” is assumed hidden at the start of mantissa, so there are a total of 53 mantissa bits.
- Approximate decimal exponent range: 10^{-308} to 10^{308}.
- Approximate decimal accuracy: 16 digits.
- Largest whole number that can be represented exactly: $2^{53} = 9.0072 \times 10^{15}$.

IEEE Accuracy Rules

° Take the exact value, and round it to the nearest floating point number (correct rounding).

° Break ties by rounding to nearest floating point number whose bottom bit is zero (rounding to nearest even).

° Other rounding options also available (up, down, towards 0).

° Don’t need exact value to do this!

  • Early implementors worried it might be too expensive, but it isn’t.

° Applies to

  • +,-,*,/, sqrt, conversion between formats.
  • rem(a,b) = remainder of a after dividing by b.
    - a = q*b + rem, q = floor(a/b)
    - cos(x) = cos(rem(x,2*pi))   for |x| >= 2*pi
    - cos(x) is *exactly* periodic, with period rounded(2*pi)

Error Analysis

° Basic error formula

- fl(a op b) = (a op b)*(1 + d) where
  - op one of +,-,*,/
  - |d| <= macheps
  - assumes no overflow, underflow, or divide by zero.

° Example: adding 4 numbers

- $fl(x_1+x_2+x_3+x_4) = \{[(x_1+x_2)*(1+d_1) + x_3]*(1+d_2) + x_4\}*(1+d_3)$

  $= x_1*(1+d_1)*(1+d_2)*(1+d_3) + x_2*(1+d_1)*(1+d_2)*(1+d_3)$

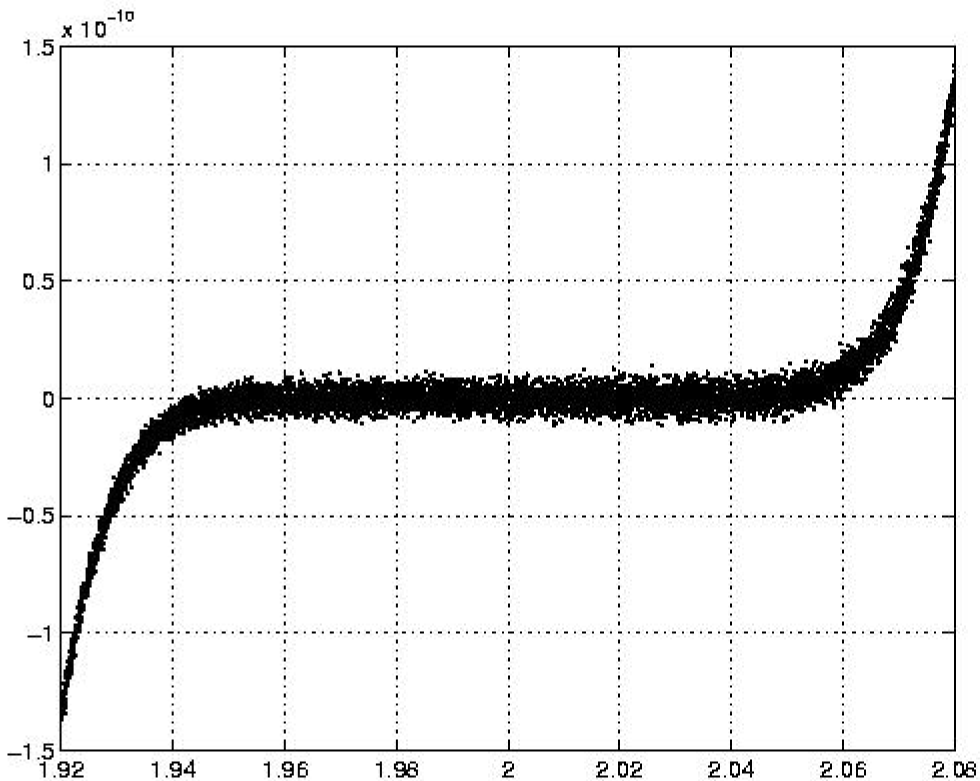
  $+ x_3*(1+d_2)*(1+d_3) + x_4*(1+d_3)$

  $= x_1*(1+e_1) + x_2*(1+e_2) + x_3*(1+e_3) + x_4*(1+e_4)$

  where each $|e_i|$ <~ 3*macheps
- Result is exact sum of slightly changed summands $x_i*(1+e_i)$.
- Backward Error Analysis - an algorithm called numerically stable if it gives the exact result for slightly changed inputs.
- Numerical stability is a fundamental algorithm design goal.

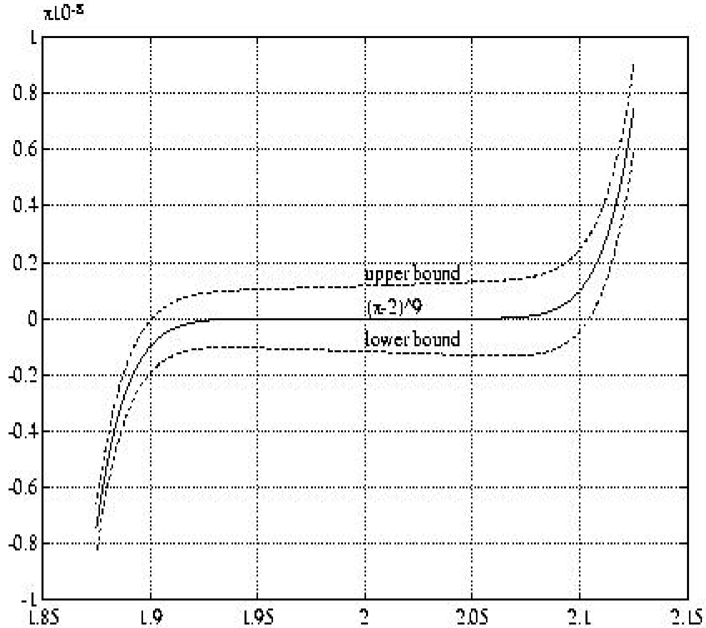
Example: Polynomial Evaluation using Horner's Rule

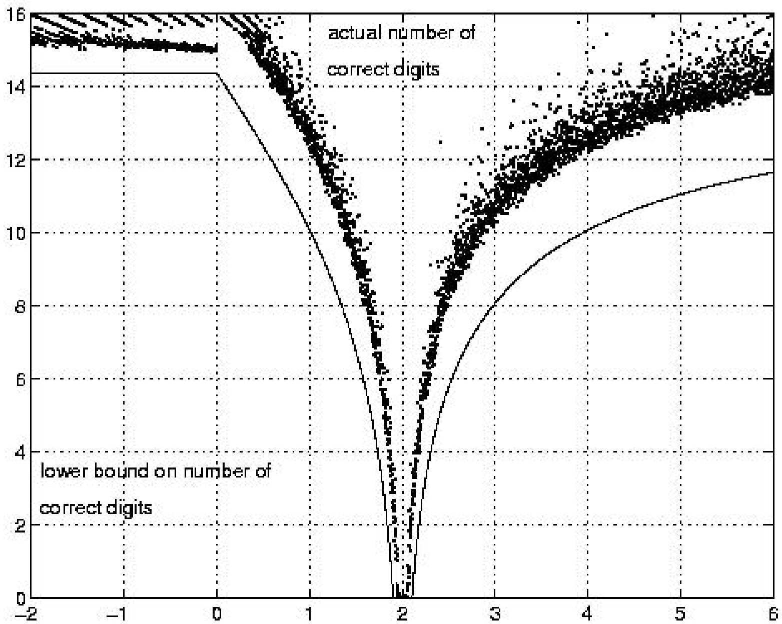
- Use Horner's rule to evaluate $p = \sum_{k=0}^{n} c_k * x^k$
  - Set $p = c_n$, then for $k=n-1$ downto 0, set $p = x*p + c_k$
- Apply to $(x-2)^9 = x^9 - 18*x^8 + \ldots - 512$.
- Error plot:



Example: Polynomial Evaluation (continued)

- $(x-2)^9 = x^9 - 18*x^8 + \ldots - 512$
- We can compute error bounds using
  - fl(a op b)=(a op b)*(1+d)





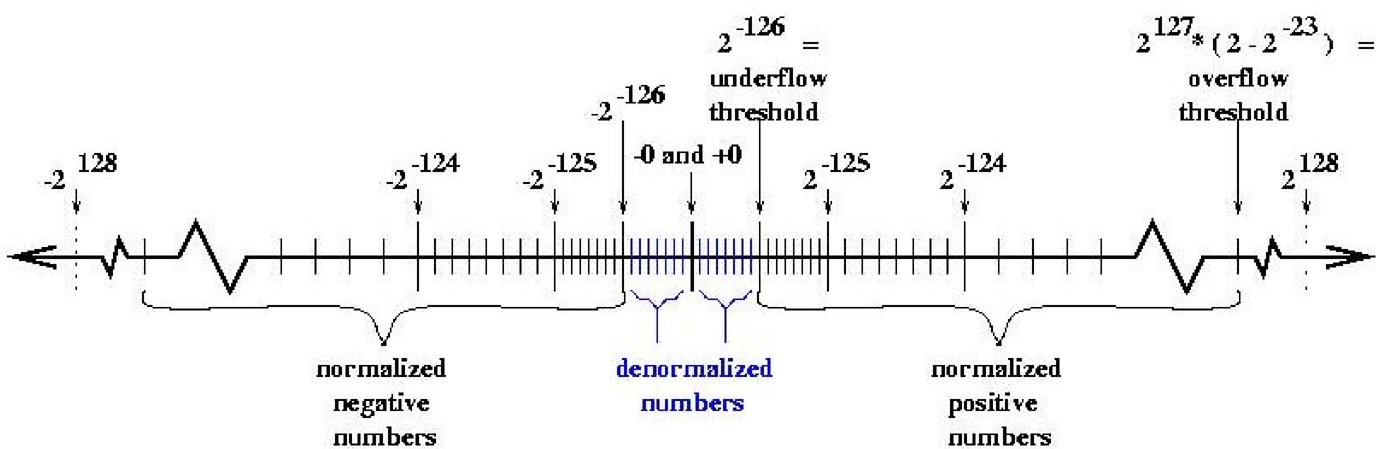
Exception Handling

- What happens when the “exact value” is not a real number, or too small or too large to represent accurately?
- Five exceptions:
  - Overflow - exact result > OV, too large to represent.
  - Underflow - exact result nonzero and < UN, too small to represent.
  - Divide-by-zero - nonzero/0.
  - Invalid - 0/0, sqrt(-1), …
  - Inexact - you made a rounding error (very common!).
- Possible responses
  - Stop with error message (unfriendly, not default).
  - Keep computing (default, but how?).

IEEE FP Arithmetic Standard 754: Denorms

- **Denormalized Numbers: +-0.d…d x 2^{min_exp}**
  - Sign bit, nonzero significand, minimum exponent.
  - Fills in gap between UN and 0.
- **Underflow Exception**
  - Occurs when exact nonzero result is less than underflow threshold UN.
  - Ex: UN/3.
  - return a denorm, or zero.
- **Why bother?**
  - Necessary so that following code never divides by zero .

    if (a != b) then x = a/(a-b)



IEEE FP Arithmetic Standard 754: +- Infinity

- +- Infinity: Sign bit, zero significand, maximum exponent.
- Overflow Exception
  - Occurs when exact finite result too large to represent accurately.
  - Ex: 2*OV.
  - return +- infinity.
- Divide by zero Exception
  - return +- infinity = 1/+-0.
  - sign of zero important!
- Also return +- infinity for
  - 3+infinity, 2*infinity, infinity*infinity.
  - Result is exact, not an exception!

IEEE FP Arithmetic Standard 754: NAN (Not A Number)

- **NAN: Sign bit, nonzero significand, maximum exponent.**
- **Invalid Exception**
  - Occurs when exact result not a well-defined real number.
  - 0/0.
  - sqrt(-1)
  - infinity-infinity, infinity/infinity, 0*infinity.
  - NAN + 3.
  - NAN > 3?
  - Return a NAN in all these cases.
- **Two kinds of NANs**
  - Quiet - propagates without raising an exception.
  - Signaling - generate an exception when touched (good for detecting uninitialized data).

Exception Handling User Interface

Each of the 5 exceptions has the following features:

- **A sticky flag, which is set as soon as an exception occurs.**
- **The sticky flag can be reset and read by the user:**
  - reset overflow_flag and invalid_flag.
  - perform a computation.
  - test overflow_flag and invalid_flag to see if any exception occurred.
- **An exception flag, which indicate whether a trap should occur:**
  - Not trapping is the default.
  - Instead, continue computing returning a NAN, infinity or denorm.
  - On a trap, there should be a user-writeable exception handler with access to the parameters of the exceptional operation.
  - Trapping or "precise interrupts" like this are rarely implemented for performance reasons.

Exploiting Exception Handling to Design Faster Algorithms

- **Paradigm:**
  1) Try fast, but possibly "risky" algorithm.
  2) Quickly test for accuracy of answer (use exception handling).
  3) In rare case of inaccuracy, rerun using slower "low risk" algorithm.
- **Quick with high probability (ie avoid branches):**
  - Assumes exception handling done quickly.
- **Ex 1: Solving triangular system Tx=b.**
  - Part of BLAS2 - highly optimized, but risky.
  - If T "nearly singular", expect very large x, so scale inside inner loop: slow but low risk.
  - Use paradigm with sticky flags to detect nearly singular T.
  - Up to 9x faster on Dec Alpha.
- **Ex 2: Computing eigenvalues, up to 1.5x faster on CM-5.**

For k= 1 to n
  $d = a_k - s - b_k^2/d$
  if $|d| < tol$, $d = -tol$
  if $d < 0$, count++

vs.

For k= 1 to n
  $d = a_k - s - b_k^2/d$ … ok to divide by 0
  count += signbit(d)

- **Demmel/Li (www.nersc.gov/~xiaoye)**

Summary of Values Representable in IEEE FP

- +- Zero
- Normalized nonzero numbers
- Denormalized numbers
- +-Infinity
- NANs
  - Signaling and quiet
  - Many systems have only quiet

+-	0…0	0……………………0
+-	Not 0 or all 1s	anything
+-	0…0	nonzero
+-	1….1	0……………………0
+-	1….1	nonzero

High Precision Arithmetic

- **What if 64 or 80 bits is not enough?**
  - **Very large problems on very large machines may need more.**
  - **Sometimes only known way to get right answer (example: mesh generation).**
  - **Sometimes you can trade communication for extra precision.**
- **High precision can be simulated efficiently using standard FP ops.**
- **Each extended precision number s is represented by an array $(s_1,s_2,\ldots,s_n)$ where:**
  - **each s_k is a FP number**
  - **$s = s_1 + s_2 + \ldots + s_n$ in exact arithmetic**
  - **$s_1 >> s_2 >> \ldots >> s_n$**
- **Ex: Computing $(s_1,s_2) = a + b$**
  - **if |a|<|b|, swap them**
  - **$s_1 = a+b$ … *roundoff may occur***
  - **$s_2 = (a - s_1) + b$ … *no roundoff!***
  - **s_1 contains leading bits of a+b, s_2 contains trailing bits**
- **Current effort to define double-double BLAS this way:**
  - **www.netlib.org/cgi-bin/checkout/blast/blast.pl**
- **Can be extended to arbitrary precision:**
  - **Priest / Shewchuk (www.cs.berkeley.edu/~jrs)**

Techniques for Very High Precision Arithmetic

- Represent data as strings of integer or FP data.
- First few words define length, sign and exponent; followed by mantissa words.
- Use standard arithmetic algorithms, but base 2^24, 2^32 or 2^53, instead of base 10. Base 100 example:

  (22, 33, 44) x (55, 66, 77) = (1210, 3267, 6292, 5445, 3388)

  = (12, 43, 30, 46, 78, 88) after release of carries starting at end.
- For very high precision (> 1000dp), use FFTs for multiplication:

  $A \times B = FFT^{-1}(FFT(A,0) \times FFT(B,0))$ (ie linear convolution)

  where (A,0) means append n words of zeroes to the n-word mantissa.
- "Quadratically convergent" algorithms (each iteration approximately doubles number of accurate digits) are known for sqrt(x), pi, e^x, cos(x), and other transcendental functions.

Bailey's High Precision Software

- **Double-double package:**
  - Double-double data is represented as pairs of 64-bit FP numbers.
  - Uses IEEE arithmetic tricks mentioned on previous slide.
  - Real and complex datatypes, also sqrt, cos, e^x, etc.
  - Declare DD variables with a Fortran-90 type statement, and the proper routines from the library are automatically called whenever any of these variables appears in an expression.
  - A C++ interface and a quad-double package are in the works.
- **Multiprecision package:**
  - Provides an arbitrarily high precision level.
  - Uses FFTs and other advanced algorithms where appropriate.
  - Fortran-90 interface permits very easy conversion of Fortran.
  - C/C++ version is also available.
- **More info: www.nersc.gov/~dhbailey**

Cray Arithmetic

- **Historically very important**
  - Crays among the fastest machines.
  - Other fast machines emulated it (Fujitsu, Hitachi, NEC).
- **Sloppy rounding**
  - fl(a + b) not necessarily (a + b)(1+d) but instead.

    $fl(a + b) = a*(1+d_a) + b*(1+d_b)$ where $|d_a|,|d_b| <= macheps$
  - This means that fl(a+b) could be either 0 when should be nonzero, or twice too large when a+b "cancels".
  - Sloppy division too.
- **Some impacts:**
  - arccos($x/\mathrm{sqrt}(x^2 + y^2)$) can yield exception, because $x/\mathrm{sqrt}(x^2 + y^2) >1$
  - mod (a, b) sometimes greater than b.
  - Best available eigenvalue algorithm fails.
    - Need $\Pi_k (a_k - b_k)$ accurately.
    - Need to preprocess by setting each $a_k = 2*a_k - a_k$ (kills bottom bit).
- **Most Cray (=SGI) systems now incorporate IEEE arithmetic.**

Hazards of Parallel and Heterogeneous Computing

What new bugs arise in parallel floating point programs?

- **Hazard #1: Non-repeatability - makes debugging very hard.**
- **Hazard #2: Different exception handling - may cause program to hang.**
- **Hazard #3: Different rounding (even on IEEE FP machines) - may result in strange errors.**

See www.netlib.org/lapack/lawns/lawn112.ps

Hazard #1: Nonrepeatability due to Nonassociativity

° Consider s= all_reduce(x,"sum") = x1 + x2 + … + xp

° Answer depends on order of FP evaluation:

- All answers differ by at most p*macheps*(|x1| + … + |xp|)
- Some orders may overflow/underflow, others not!

° How can order of evaluation change?

- Change number of processors.
- In reduction tree, have each node add first available child sum to its own value - order of evaluation depends on race condition, which is unpredictable!

° Options

- Live with it, since difference likely to be small.
- Build slower version of all_reduce that guarantees evaluation order independent of #processors, to use for debugging.
- Use double-double arithmetic to guarantee repeatable answer -- see He/Ding paper, "Using Accurate Arithmetics to improve Numerical Reproducibility and Stability in Parallel Applications". URL: http://www.nersc.gov/research/SCG/ocean/NRS

Hazard #2: Heterogeneity: Different Exception Defaults

° **Not all processors implement denorms fast:**

- **DEC Alpha 21164 in "fast mode" flushes denorms to zero:**
  - **in fast mode, a denorm operand causes a trap.**
  - **slow mode, to get underflow right, slows down all operations significantly, so rarely used.**
- **SUN Ultrasparc in "fast mode" handles denorms correctly:**
  - **handles underflow correctly at full speed.**
  - **flushing denorms to zero requires trapping, slow.**

° **Imagine a NOW built of DEC Alphas and SUN Ultrasparcs:**

- **Suppose the SUN sends a message to a DEC containing a denorm: the DEC will trap.**
- **Avoiding trapping requires running either DEC or SUN in slow mode.**
- **Good news: most machines are converging to fast and correct underflow handling.**

Hazard #3: Heterogeneity: Data Dependent Branches

° Mixed Cray/IEEE machines may round differently.

° Different "IEEE machines" may round differently:

- Intel uses 80 bit FP registers for intermediate computations
- IBM RS6K has MAC = Multiply-ACcumulate instruction
  - d = a*b+c with one rounding error, i.e. a*b good to 104 bits
- SUN has neither "extra precise" feature.
- Different compiler optimizations may round differently (yuck).

° Impact: same expression can yield different values on different machines.

```
Compute s redundantly
   or
s = reduce_all(x,min)
if (s > 0) then
     compute and return a
else
     communicate
     compute and return b
end
```

° Taking different branches can yield nonsense, or deadlock.

Further References on Floating Point Arithmetic

° Notes for Prof. Kahan's CS267 lecture from 1996

- www.cs.berkeley.edu/~wkahan/ieee754status/cs267fp.ps
- Note for Kahan 1996 cs267 Lecture

° Prof. Kahan's "Lecture Notes on IEEE 754"

- www.cs.berkeley.edu/~wkahan/ieeestatus/ieee754.ps

° Prof. Kahan's "The Baleful Effects of Computer Benchmarks on Applied Math, Physics and Chemistry

- www.cs.berkeley/~wkahan/ieee754status/baleful.ps

° Notes for Demmel's CS267 lecture from 1995

- www.cs.berkeley.edu/~demmel/cs267/lecture21/lecture21.html